LISA สูตรลับการเขียนภาษาแอสเซมบลีให้เป็นของง่าย ตอน 2

ณรงค์พร เหล่าศรีสิน

ในฉบับที่ 26 ได้กล่าวถึงลิซ่าในการเขียนโปรแกรมแอสเซมบลีไปแล้ว แต่เพื่อให้คุณผู้อ่านได้ใช้งานลิซ่าอย่างมีประสิทธิภาพ ผมจึงได้เตรียมหัวข้อที่น่าสนใจให้คุณได้อ่านกันอีก 3 หัวข้อ คือ

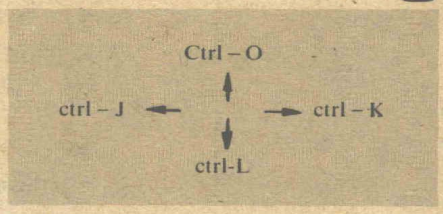
1. การใช้งานอีดิตเตอร์ของลิซ่าในการเขียนโปรแกรม
2. การจัดพื้นที่หน่วยความจำของโปรแกรมลิซ่า
3. เรื่องราวน่าสนใจเกี่ยวกับบางคำสั่งของ 6502

การใช้งานอีดิตเตอร์ของลิซ่า

เราสามารถใช้อีดิตเตอร์ของลิซ่าในการเขียนโปรแกรมได้ง่าย ทั้งนี้เนื่องจากมีปุ่มควบคุมเพียง 4 ปุ่มเท่านั้น แต่ก็นับว่าเป็นอีดิตเตอร์ที่มีประสิทธิภาพมากตัวหนึ่งเหมือนกัน เพราะเป็นอีดิตเตอร์แบบฟูลสกรีน ทำให้เราสามารถโยกย้ายตำแหน่งของเคอร์เซอร์ไปยังส่วนใดก็ได้บนจอภาพ นอกจากนี้ยังมีความสามารถในการลบหรือแทรกข้อมูลต่าง ๆ สามารถก๊อบปี้หรือโยกย้ายข้อความที่ต้องการไปยังตำแหน่งใดของโปรแกรมก็ได้ด้วย

ปุ่มที่ใช้ควบคุมการย้ายตำแหน่งเคอร์เซอร์มีอยู่ 4 ปุ่ม ดังรูปที่ 9

9



ctrl-O ใช้เลื่อนเคอร์เซอร์ขึ้น

ctrl-L ใช้เลื่อนเคอร์เซอร์ลง

ctrl-J ใช้เลื่อนเคอร์เซอร์ไปทางซ้าย

ctrl-K ใช้เลื่อนเคอร์เซอร์ไปทางขวา

ปุ่มควบคุมทั้ง 4 ปุ่มนี้ ใช้ในการเคลื่อนย้ายเคอร์เซอร์ไปยังตำแหน่งที่ต้องการเท่านั้น โดยจะไม่สนใจตัวอักษรต่าง ๆ ซึ่งเคอร์เซอร์เคลื่อนที่ผ่านไป และจะไม่นำเอาตัวอักษรเหล่านั้นไปเก็บไว้ในคีย์บอร์ดบัฟเฟอร์เลย ซึ่งนี่เป็นลักษณะที่ทำให้การเลื่อนเคอร์เซอร์ด้วยการกดปุ่ม ctrl-J หรือ ctrl-K ได้ผลลัพธ์แตกต่างจากการใช้ปุ่มลูกศรซ้ายหรือขวา คือเมื่อใช้ปุ่มลูกศรชี้ซ้าย–ขวาผ่านตัวอักษรใด ลิซ่าก็จะนำตัวอักษรเหล่านี้เข้าไปเก็บไว้ในคีย์บอร์ดบัฟเฟอร์ด้วย

โดยอาศัยเพียงลักษณะหน้าที่ของคีย์บอร์ดทั้ง 6 ปุ่มนี้ เราก็สามารถใช้แก้ไขโปรแกรมได้แล้ว ลักษณะการแก้ไขโปรแกรมมีอยู่ 4 แบบด้วยกันคือ

- การแทรกข้อความเพิ่มเติม
- การลบข้อความที่ไม่ต้องการ
- การก๊อบปี้ข้อความที่ต้องการ
- การเคลื่อนย้ายบางส่วนของโปรแกรมไปไว้ยังตำแหน่งใหม่

การแทรกข้อความ ผมขอยกตัวอย่างการแทรกข้อความประกอบคำอธิบายไว้ดังนี้ สมมติว่า ผมต้องการเปลี่ยนโปรแกรมภาษาแอสเซมบลีในบรรทัดที่ 7 จากเดิม HEX 0079AB ไปเป็น HEX 006593FF79AB ตัวเลขที่ผมขีดเส้นใต้เอาไว้คือตัวเลขที่ผมต้องการแทรกลงไปเพิ่มเติม ขั้นตอนการแทรกข้อความมีลำดับดังต่อไปนี้ คือ

1. ใช้คำสั่ง MODIFY เพื่อแก้ไขโปรแกรมในบรรทัดที่ 7 ดังรูปที่ 10

10

```
! M 7                  return
  7                    H EX 0079AB
(1) 7                  เคอร์เซอร์ (2)
```

2. กดปุ่ม ctrl-O เพื่อเลื่อนเคอร์เซอร์ขึ้นไปหนึ่งบรรทัดไปยังตำแหน่งที่ (1) แล้วกดปุ่ม → เลื่อนเคอร์เซอร์ไปยังตำแหน่งที่ (2) จากนั้นเลื่อนเคอร์เซอร์ขึ้นข้างบนหนึ่งบรรทัดด้วยปุ่ม ctrl-O แล้วจึงพิมพ์ข้อความ 6593FF ลงไป เมื่อพิมพ์เสร็จให้ใช้ปุ่ม ctrl-J และ ctrl-L เลื่อนเคอร์เซอร์มาอยู่ที่ตำแหน่ง (2) ดังเดิม จากนั้นจึงใช้ → เลื่อนเคอร์เซอร์ไปสุดบรรทัดตามด้วยปุ่ม return ก็เป็นอันเสร็จพิธีการ

เป็นอย่างไรบ้างครับ สำหรับการแทรกข้อความด้วยวิธีนี้ ไม่ยากเลยใช่ไหมครับ

การลบข้อความ ผมก็ขี้เกียจจะยกตัวอย่างใหม่ ขอใช้ตัวอย่างเก่าก็แล้วกัน แต่ความต้องการของผมตอนนี้คือต้องการลบข้อความ 6593FF ออกไปให้เหลือเพียง HEX 0079AB ดังเดิม สำหรับการแก้ไขในกรณีนี้ ทำได้โดยมีลำดับ

ขั้นตอนดังนี้

1. ใช้คำสั่ง MODIFY แก้ไขโปรแกรมในบรรทัดที่ 7 ดังรูปที่ 11

11

```
! M 7 [return]        ①      ②
   7 HEX 00[6]5 93 FF[7]9 AB
   7 [เคอร์เซอร์]
```

2. ใช้ปุ่ม ctrl-O เลื่อนเคอร์เซอร์ขึ้นไป 1 บรรทัด แล้วใช้ปุ่ม → เลื่อนเคอร์เซอร์ไปตำแหน่งที่ ① ต่อไปเปลี่ยนไปใช้ปุ่ม ctrl-K เลื่อนไปยังตำแหน่งที่ ② แล้วกลับมาใช้ปุ่ม → เลื่อนเคอร์-เซอร์ไปที่สุดบรรทัด ตบท้ายด้วยปุ่ม return ตามระเบียบ ก็เป็นอันว่าจบการแก้ไขแล้ว

การที่เราใช้ปุ่ม ctrl-K เลื่อนเคอร์-เซอร์จากตำแหน่ง ① ไปยังตำแหน่ง ② ก็จะทำให้ไม่มีการรับตัวอักษรใด ๆ เข้าไปเก็บในคีย์บอร์ดบัฟเฟอร์เลย ดังนั้นข้อความ 6593FF จึงหายไป เมื่อเราใช้คำสั่ง list ก็จะพบกับข้อความ HEX 0079AB ตามต้องการ

การก๊อบปี้ข้อความ สมมติว่าคุณต้องการก๊อบปี้โปรแกรม บรรทัดที่ 2 และ 3 ไปไว้ยังบรรทัดที่ 11 และ 12 ตามลำดับ เราก็มีขั้นตอนการคัดลอกดังนี้คือ

1. ลิสต์โปรแกรมเฉพาะบรรทัดที่เราต้องการก๊อบปี้ออกมา คือโปรแกรมบรรทัดที่ 2 และ 3

2. จากนั้นใช้คำสั่งแทรกโปรแกรมตรงบรรทัดที่เราต้องการ โดยป้อนคำสั่ง I 11

3. ใช้ปุ่ม ctrl-O เลื่อนเคอร์เซอร์ขึ้นจากตำแหน่งเดิมไปยังโปรแกรมในบรรทัดที่ 2 แล้วใช้ปุ่มลูกศรขวาเลื่อนเคอร์เซอร์ไปจนสุดบรรทัด แล้วจึงกดปุ่ม return ก็เป็นอันว่าเราได้ก๊อบปี้โปรแกรมจากบรรทัดที่ 2 มาไว้ยังบรรทัดที่ 11 เรียบร้อยแล้ว

4. มาถึงตอนนี้แล้ว คุณคงสังเกตเห็นว่าตัวเลขบรรทัดที่ 3 ขณะนี้เปลี่ยนเป็นตัวเลข 12 แล้ว เราเลื่อนเคอร์เซอร์ไปจนสุดบรรทัดด้วยปุ่มลูกศรขวา แล้วตามด้วยปุ่ม return เหมือนเช่นเคย จากนั้นก็จบการแก้ไขโดยกดปุ่ม ctrl-E และ return เพื่อออกจากโหมดการแทรกโปรแกรมได้เลย

การเคลื่อนย้ายโปรแกรม สำหรับการเคลื่อนย้ายบางส่วนของโปรแกรมนั้นเราก็ทำโดยการก๊อบปี้โปรแกรมส่วนนั้นไปยังตำแหน่งใหม่ที่เราต้องการ แล้วจึงใช้คำสั่งลบโปรแกรมที่ตำแหน่งเดิมทิ้งไป ก็จะเท่ากับเราได้เคลื่อนย้ายโปรแกรมเป็นที่เรียบร้อยแล้ว

การจัดพื้นที่หน่วยความจำของลิซ่า

ที่ผมจะกล่าวถึงต่อไปนี้เป็นเรื่องของการจัดพื้นที่หน่วยความจำของโปร-แกรมลิซ่า ซึ่งแบ่งไว้ใช้งาน ดังนี้

แอดเดรส 50-AF และ E0-FF ลิซ่าใช้เก็บค่าตัวแปรต่างๆ ในโปรแกรม

800-1800 เป็นส่วนที่ใช้เก็บออบเจคโค้ดที่ได้จากการแปล

1800-7FFF ใช้เก็บโปรแกรมภาษาแอสเซมบลีที่เราเขียนขึ้น(text file)

8000-94FF เป็นส่วนที่ใช้เก็บค่าของเลเบลและสัญลักษณ์ต่าง ๆ

9500-95FF เป็นส่วนของ I/O BUFFER ของลิซ่า

D000-F7FF เป็นที่ตั้งของโปร-แกรมลิซ่าเอง

นอกจากเรื่องการแบ่งส่วนหน่วยความจำเพื่อใช้งานโดยเฉพาะแล้ว ผมก็อยากจะกล่าวถึงแอดเดรสสำคัญต่างๆ ของลิซ่าด้วย

$E000 เป็นจุดเริ่มต้นของการเข้าสู่ลิซ่า เช่น เมื่อคุณอยู่ในโปรแกรมมอนิเตอร์ก็สามารถเข้าสู่ลิซ่าได้ด้วยคำสั่ง $E000G หรือโดยการกดปุ่ม ctrl-B ก็ได้ ผลจากคำสั่งนี้ ลิซ่าก็จะเคลียร์โปรแกรมภาษาแอสเซมบลีเดิมที่มีอยู่ในเครื่องให้หายไปเหมือนกับการใช้คำสั่ง NEW อย่างไงอย่างงั้นเลย เราเรียกการเริ่มรันลิซ่าด้วยวิธีนี้ว่า coldstart

$E003 เป็นแอดเดรสที่เราจะเข้าสู่ลิซ่าได้อีกทางหนึ่ง แต่การใช้คำสั่ง E003G หรือการกดปุ่ม ctrl-C จะทำให้สามารถเข้าสู่ลิซ่าได้ โดยที่โปรแกรมภาษาแอสเซมบลีเดิมยังคงอยู่ในเครื่อง การเข้าสู่ลิซ่าในลักษณะนี้เรียกว่า warm-start

$E009 เป็นที่เก็บคำสั่งกระโดดไปยังโปรแกรมภาษาเครื่องที่คุณเขียนขึ้น โดยเมื่อมีการกดปุ่ม ctrl-P แทนการป้อนคำสั่งต่างๆ ก็จะทำให้ลิซ่ากระโดดมายังที่แอดเดรส E009 และที่ตำแหน่งนี้ลิซ่าก็จะถูกสั่งให้กระโดดไปยังโปรแกรมที่คุณต้องการอีกทีหนึ่ง

$E01B ลิซ่าใช้หน่วยความจำแอดเดรสนี้สำหรับเก็บค่าของแอดเดรสไบท์สูง ที่เป็นตำแหน่งเริ่มต้นของโปรแกรมภาษาแอสเซมบลี ซึ่งโดยปกติแล้วลิซ่าจะเก็บค่า $18 ไว้ที่ไบท์นี้

$E01C ใช้สำหรับเก็บค่าแอดเดรสไบท์สูงของตำแหน่งเริ่มต้นของ SYMBOL TABLE หรือตำแหน่งสุดท้ายของที่เก็บโปรแกรมแอสเซมบลี ซึ่งโดยทั่วไปจะบรรจุค่า $80 เอาไว้

สำหรับค่าตัวเลขในตำแหน่ง E01B และ E01C นั้น เราสามารถเปลี่ยนแปลงได้ ซึ่งก็เท่ากับเป็นการเปลี่ยนแปลงขนาดของส่วนที่เก็บโปรแกรมภาษาเครื่อง และโปรแกรมภาษาแอสเซมบลี และเกี่ยวข้องกับขนาดของ SYMBOL TABLE ด้วย การเปลี่ยนแปลงค่าต่างๆ เหล่านี้ ควรกระทำด้วยความระมัดระวังนะครับ

อ้อ! ถ้าหากคุณมีเครื่องแอปเปิ้ลทูขนาด 48K ล่ะก็ อาจจะต้องมีการเปลี่ยนแปลงค่าแอดเดรสต่างๆ เพราะที่กำหนดไว้ข้างต้นนั้นเป็นกรณีของเครื่องที่มีแรม

64K เมื่อจำนวนหน่วยความจำลดลง ลิ–ซ่าก็จำเป็นจะต้องลดพื้นที่ต่างๆ ลงด้วย รวมทั้งเปลี่ยนแปลงค่าแอดเดรสที่สำคัญต่างๆ

สำหรับค่าแอดเดรสที่เก็บค่าตัวแปรต่างๆ หรือแอดเดรสตั้งแต่ $E000 จนถึง E01C นั้น ในกรณีที่มีแรม 48K ก็ให้เอาตัวเลข $8000 ไปลบออกจากค่าแอดเดรสที่กำหนดไว้ ก็จะได้ค่าแอดเดรสที่ถูกต้องอย่างเช่น การเข้าสู่ลิซ่านั้นยังคงทำได้สองทางคือ การป้อนคำสั่ง $6000G หรือ $6003G ก็ได้ แล้วแต่ว่าคุณต้องการ coldstart หรือ warmstart แต่การกดปุ่ม ctrl-B หรือ ctrl-C ไม่สามารถใช้กับลิซ่า 48K ได้

นอกจากนี้ขนาดของหน่วยความจำที่เก็บโปรแกรมภาษาแอสเซมบลีนั้น จำเป็นต้องลดขนาดลงด้วย ซึ่งถ้าหากคุณพิจารณาค่าตัวเลขที่แอดเดรส 601B และ 601C ก็จะพบว่า ขอบเขตที่เก็บโปรแกรมมีตั้งแต่ $1800-5800 แทนที่จะเป็น 1800-8000

อ่านมาถึงตอนนี้แล้วก็ไม่ทราบว่าคุณๆ ผู้อ่านเบื่อกันหรือยังเพราะผมเขียนมาซะยาวเลย กะเอาไว้ว่าจะให้จบในตอนเดียวเล่มเดียว เพื่อว่าคุณอาจจะใช้วารสารไมโครฯเล่มนี้ แทนคู่มือการใช้ลิซ่าเลยก็ได้ เมื่อเวลามีปัญหาเกิดขึ้น

เก็บตกจากชุดคำสั่งของ 6502

เรื่องสุดท้ายนี้ไม่ได้เกี่ยวข้องอะไรกับลิซ่าหรอกครับ แต่เป็นเรื่องเล็กๆน้อยๆ ที่ผมอยากจะเล่าให้ฟังเกี่ยวกับชุดคำสั่งของ 6502 คุณผู้อ่านยังจำได้ไหมว่า มีตารางสรุปชุดคำสั่งของ 6502 อยู่ในบทความเรื่อง "สนุกกับภาษาแอสเซมบลี" วารสารไมโครฯ ฉบับที่ 14 ผมคิดว่าหลายต่อหลายคนคงจะสามารถทำความเข้าใจคำสั่งต่างๆ ในนั้นได้ไม่ยากนักใช่ไหมครับ เพราะดูเหมือนว่า ตารางดังกล่าวมีรายละเอียดที่คุณควรจะทราบอยู่เกือบครบแล้ว ครับ! ผมพูดว่า "เกือบครบ" ก็เพราะว่ายังมีบางส่วนตกหล่นอยู่ และผมก็กำลังจะเก็บมาเล่าให้ฟังอยู่นี่แหละครับ เป็นต้นว่า เรื่องเกี่ยวกับ.....

– การบวก–ลบตัวเลขในโหมดเลขฐานสิบ

– การเปรียบเทียบดูว่า ใครจะมีค่ามากกว่าใคร

– การกระโดดโดยไม่ต้อง JUMP

– และการใช้คำสั่ง BIT ตรวจสอบข้อมูลในหนึ่งไบท์

การคำนวณตัวเลขใน DECIMAL MODE หรือการคำนวณตัวเลขฐานสิบนั้น ก็จะเหมือนกับการคำนวณตัวเลขฐานสิบหก เพียงแต่ตัวเลขที่ใช้จะเป็นเลข 0–9 แทนที่จะเป็น 0–F

โดยปกติการบวกตัวเลขจะอยู่ในโหมดเลขฐาน 16 แต่เมื่อเราต้องการคำนวณเลขฐานสิบ เราจะใช้คำสั่ง SED เพื่อเซ็ท DECIMAL MODE และเซ็ทแฟล็ก D ให้เป็นหนึ่งด้วย แต่เมื่อต้องการยกเลิก DECIMAL MODE ก็ใช้คำสั่ง CLD แฟล็ก D ก็จะกลับเป็นศูนย์ และเข้าสู่โหมดเลขฐานสิบหกเช่นเดิม ลองดูตัวอย่างโปรแกรมในรูปที่ 12 หน่อยเป็นไง

12

```
CLC
SED
LDA #$27
ADC #$53
CLD
```

ผลลัพธ์ที่ได้จากการบวกเลขนี้ก็คือ $80 ก็เป็นอันว่าการบวกเลขฐานสิบ ไม่มีปัญหาอะไรเท่าไหร่ แต่ถ้าผมเปลี่ยนตัวเลขจาก $27 ไปเป็น $2A ล่ะ ผลลัพธ์จะเป็นอย่างไร เดาออกไหมครับ แต่ถึงอย่างไรผมก็ต้องเฉลยอยู่ดีละว่า ผลลัพธ์ของการบวกจะกลายเป็น $83 นั่นคือตัวเลข $2A นั้นจะมีความหมายเท่ากับ $30 ไปเลย

การบวกตัวเลขดูท่าจะไม่มีปัญหาอะไรมาก แต่เรื่องการลบเลขนี่ซิ ยิ่งเป็นการลบตัวเลขในโหมดเลขฐานสิบก็ยิ่งมีอะไรให้คุณฉงนสงสัยเล่น ดังเช่นตัวอย่างในรูปที่ 13 นี้ คุณลองเดาผลลัพธ์ของการลบเลข 2 ค่านี้ดูซิครับ

13

```
SEC
SED
LDA #$45
SBC #$73
CLD
```

ผลลัพธ์ คือ $72 ครับ และถ้าหากคุณทดลองเปลี่ยนค่าตัวเลขดู จะพบว่า

$45-$98 = $47
$0 -$98 = $2
$47-$22 = $25

หลายคนอาจจะทราบแล้วว่าทำไมค่าผลต่างจึงออกมาเป็นเช่นนี้ ทั้งหมดนี้เนื่องมาจากการเปลี่ยนตัวลบไปเป็นเลข 2 คอมพลีเมนท์ แต่เนื่องจากอยู่ในโหมดเลขฐานสิบ ดังนั้นการเปลี่ยนเป็นเลข 2 คอมพลีเมนท์จึงได้ค่าไม่เหมือนกับในโหมดเลขฐานสิบหก เรามีวิธีการคำนวณดังนี้

เลข 2 คอมพลีเมนท์ของ X คือ 100-X

ดังนั้นการหาค่าของการลบเลข $45-$73 ในโหมดเลขฐานสิบ จึงมีขั้นตอนดังนี้

– คำนวณเลข 2 คอมพลีเมนท์ของ 73 ซึ่งมีค่าเท่ากับ 100–73 = 27

– นำมาบวกกับตัวตั้ง 45+27 = 72

– เนื่องจากค่าผลลัพธ์ไม่เกิน 100 ดังนั้นแฟล็กตัวทดจึงเป็นศูนย์ ซึ่งแสดงว่าผลลัพธ์ที่ได้เป็นเลขลบ 2 คอม–พลีเมนท์ ซึ่งถ้าหากเราเปลี่ยนผลลัพธ์นี้ไปเป็นเลขบวกธรรมดา ก็จะได้ 100–72 = 28 นั่นคือ 45–73 = –28

แต่ถ้าหากเราเอา 47–22 จะได้ 25 แต่แฟล็กตัวทด = 1 ก็หมายความว่าผลลัพธ์ที่ได้เป็นค่าบวก

การเปรียบเทียบค่าตัวเลข คำสั่งเกี่ยวกับการเปรียบเทียบอันได้แก่ คำสั่ง CMP, CPX และ CPY เป็นคำสั่งที่ใช้เปรียบเทียบค่าในรีจิสเตอร์กับค่าในหน่วยความจำ ผลจากการเปรียบเทียบจะปรากฏอยู่ที่แฟล็กต่างๆ ในสเตตัสรีจิสเตอร์ เราลองมาดูการเปรียบเทียบในกรณีต่างๆ เหล่านี้ และผลที่มีต่อแฟล็กต่างๆ ดังตารางในรูปที่ 14 โดยเป็นการเปรียบเทียบค่าในรีจิสเตอร์ (A) กับค่าในหน่วยความจำ (B)

14

แฟล็ก	C	Z	N*
A มากกว่า B	1	0	0
A เท่ากับ B	1	1	0
A น้อยกว่า B	0	0	1

อาจจะมีหลายคนตั้งคำถามขึ้นมาว่าทำไมผลจากการเปรียบเทียบจึงทำให้แฟล็กมีค่าเช่นนี้ ผมก็จะเริ่มอธิบายกันตั้งแต่ว่า.....

เมื่อมีการเปรียบเทียบค่าตัวเลขนั้น ซีพียูจะนำตัวเลขสองตัวนั้นมาลบกัน แต่จะเป็นการลบค่าภายในซีพียูเท่านั้น ค่าในรีจิสเตอร์กับค่าในหน่วยความจำยังคงเหมือนเดิมไม่เปลี่ยนแปลง ในการลบตัวเลขฐานสิบหกนั้นมีหลักอยู่ว่า จะต้องเปลี่ยนตัวลบเป็นเลข 2 คอมพลีเมนท์แล้วนำไปบวกรวมกับตัวตั้ง ก็จะได้ผลต่างออกมา (แต่เราไม่สนใจค่าผลต่างนั้นหรอก) และค่าแฟล็กตัวทดจะเป็นตัวกำหนดเครื่องหมายบวก ลบ ของผลลัพธ์นั้น โดยถ้าหากว่า แฟล็กตัวทดเป็นหนึ่งก็แสดงว่าค่าผลลัพธ์เป็นเลขบวก หรือเลขตัวตั้งมีค่ามากกว่าเลขตัวลบ ในทางตรงกันข้าม ถ้าแฟล็กตัวทดเป็นศูนย์ก็จะหมายถึงค่าผลลัพธ์ที่ได้เป็นค่าลบ หรือหมายความว่าเลขตั้งมีค่าน้อยกว่าตัวเลขที่นำมาเปรียบเทียบด้วยนั่นเอง

ที่นี้ ถ้าหากเลขทั้งสองเท่ากัน เมื่อนำมาลบกันผลลัพธ์ก็ย่อมเป็นศูนย์ ก็จะทำให้แฟล็ก Z ถูกเซ็ทเป็นหนึ่ง ดังตารางที่ให้ไว้ ส่วนแฟล็ก N นั้นจะคอยตรวจสอบดูว่าบิทที่ 7 ของผลลัพธ์จากการลบตัวเลขนั้นมีค่าเป็น 1 หรือเป็น 0 ถ้าบิทที่ 7 ของผลลัพธ์เป็น 1 ก็จะถือว่าเป็นเลขลบและแฟล็ก N จะถูกเซทให้เป็นหนึ่งด้วย

ผมมีตัวอย่างการเปรียบเทียบอยู่อันหนึ่งที่อยากจะให้คุณได้พิจารณาดูกัน ดังตัวอย่างในรูปที่ 15

15

```
LDA #$A0
CMP #$10
BPL OUT
```

จากตัวอย่างนี้ หลายคนคงจะคิดว่าจะต้องมีการกระโดดเกิดขึ้นแน่นอน แต่ความจริงที่ปรากฏนั้น ย่อมแน่นอนกว่า ถ้าหากคุณทดลองรันโปรแกรมจริงๆ จะพบว่าไม่มีการกระโดดเกิดขึ้น แต่ถ้าคุณลองเปลี่ยนคำสั่งจาก BPL ไปเป็น BCS ล่ะก็ รับรองว่ามีการกระโดดแน่ๆ

มาถึงตอนนี้ คุณผู้อ่านก็อย่าเพิ่งหาว่าตารางที่ให้ไว้ข้างบนนั้นผิดพลาดนะครับ แต่ที่ผมยกตัวอย่างมานี้ เพื่อจะบอกคุณว่า เราไม่สามารถจะใช้แฟล็ก N ในการบอกผลเปรียบเทียบได้ทุกกรณี อย่างโปรแกรมที่ให้มานี้ คุณพอจะทราบไหมว่าเป็นเพราะสาเหตุใด แฟล็ก N จึงไม่เป็นศูนย์ตามตารางข้างบน

ผมอยากจะให้คุณลองลบตัวเลข $A0-$10 ซึ่งจะได้ผลลัพธ์เป็น $90 และแฟล็กตัวทดเป็นหนึ่ง คุณคงเห็นแล้วว่าบิทที่ 7 ของผลลัพธ์ถูกเซ็ทให้เป็นหนึ่ง ดังนั้นแฟล็ก N จึงกลายเป็นหนึ่งตามไปด้วย

สรุปแล้ว ถ้าหากคุณใช้แฟล็ก N เป็นตัวแสดงผลการเปรียบเทียบแล้ว คุณควรจะแน่ใจเสียก่อนว่า ค่าตัวเลขทั้งสองที่มาเปรียบเทียบกัน มีค่าต่างกันน้อยกว่า $80 ก็เป็นอันเรียบร้อย

การใช้คำสั่ง BIT คำสั่ง BIT เป็นคำสั่งหนึ่งที่น่าสนใจ แต่ไม่ค่อยจะได้ใช้กันนัก คงอาจเป็นเพราะยังไม่ค่อยรู้จักคำสั่งนี้ดีนัก กลัวว่าจะใช้กันผิดๆ ถูกๆ กระมัง

ถ้าหากคุณได้เปิดดูตารางสรุปคำสั่งในวารสารไมโครฯ ฉบับที่ 14 นั้น คุณก็คงจะได้อ่านคำอธิบายเกี่ยวกับคำสั่ง BIT ที่เขียนเอาไว้ว่า

"TEST BIT IN MEMORT WITH ACCUMULATOR", "AAM, $M_7 \rightarrow N$, $M_6 \rightarrow V$"

เครื่องหมาย Λ ที่เห็นนั้น หมายถึงการ AND ลอจิก ดังนั้นผู้อ่านบางคนที่ได้อ่านข้อความข้างต้นนี้ก็อาจคิดไปว่าเมื่อใช้คำสั่ง BIT แล้ว จะมีการ AND กันระหว่างข้อมูลในแอคคิวมูเลเตอร์กับข้อมูลในหน่วยความจำ แล้วนำเฉพาะบิทที่ 6 และ 7 ของผลลัพธ์ไปใส่ไว้ในแฟล็ก N และ V ตามลำดับ

แต่ในความจริง ค่าตัวเลขในแอคคิวมูเลเตอร์จะไม่เกี่ยวข้องกับคำสั่ง BIT เลย และไม่มีการนำค่าในแอคคิวมูเลเตอร์ไป AND กับค่าตัวเลขในหน่วยความจำด้วย ดังนั้น ค่าตัวเลขต่างๆ จึงยังคงเดิมอยู่ จะเปลี่ยนแปลงก็แต่เฉพาะค่าของแฟล็กเท่านั้น คือแฟล็ก N และ V จะมีการเปลี่ยนแปลงตามข้อมูลในบิทที่ 7 และ 6 ของหน่วยความจำ นอกจากนี้ ยังมีผลต่อแฟล็ก Z ด้วย ในกรณีที่ค่าในหน่วยความจำเป็นศูนย์หรือไม่เป็นศูนย์

การกระโดดโดยไม่ต้อง JUMP มันไม่ได้มีเพียงคำว่า JUMP คำเดียวเท่า

นั้น ที่มีความหมายถึงการกระโดดไปยังค่าแอดเดรสใหม่ เพราะตอนนี้ผมได้เตรียมท่ากระโดดท่าใหม่ๆ มาให้คุณได้ลองฝึกกระโดดกันแล้ว คุณพอจะนึกออกไหมว่า ท่าใหม่ๆ ที่ผมพูดถึงนั้นมีลักษณะอย่างไร ผมจะเริ่มแนะนำเลยนะครับ

1. การกระโดดแบบสัมพัทธ์(JUMP RELATIVE) สำหรับคุณๆ ที่เคยสัมผัสกับ Z80 มาแล้ว ก็คงพอจะรู้จักกับคำสั่ง กระโดดแบบสัมพัทธ์ หรือคำสั่ง JR กันดีแล้ว เมื่อซีพียูพบกับคำสั่ง JR ก็จะนำค่าของตัวเลขที่ตามท้ายคำสั่งไปรวมกับค่าของโปรแกรมเคาน์เตอร์ ทำให้ได้ค่าแอดเดรสใหม่ที่จะกระโดดไป

ในซีพียู'6502 นั้น ถึงแม้จะไม่มีคำสั่ง JR จริงๆ แต่ก็มีคำสั่งซึ่งคล้ายกัน คือ คำสั่ง BRANCH ซึ่งถ้าหากเราพิจารณาดู ก็จะพบว่าคำสั่งนี้มีลักษณะคล้ายกับคำสั่ง JR แต่มีการตรวจสอบเงื่อนไขเสียก่อน ซึ่งถ้าหากเงื่อนไขเป็นจริง ก็จะมีการกระโดด หรือการ BRANCH ไป แต่ถ้าเงื่อนไขเป็นเท็จ ก็จะทำตามคำสั่งที่อยู่ถัดไป

เรื่องเหล่านี้ หลายคนอาจจะทราบกันบ้างแล้ว พูดมากไปก็เท่านั้นแหละ อยากจะให้ดูโปรแกรมตัวอย่างในรูปที่ 16 กันเลยดีกว่า

16

```
FE80    LDY    #$3F
FE82    BNE    $FE86
FE85    LDY    #$FF
FE87    STY    $32
FE89    RTS
```

โปรแกรมนี้ เป็นเพียงบางส่วนจากโปรแกรมมอนิเตอร์ในเครื่องแอปเปิ้ล ผมอยากให้คุณสังเกตการใช้คำสั่ง BRANCH ในแอดเดรส $FE82 เราเรียกการใช้คำสั่ง BRANCH ในลักษณะนี้ว่า ALWAYS BRANCH ซึ่งก็หมายความว่า เมื่อมีการตรวจสอบเงื่อนไขที่กำหนด จะพบว่าเงื่อนไขเป็นจริงเสมอ ทำให้มีการกระโดดเกิดขึ้นเสมอ

ดังนั้นเราจึงอาจใช้คำสั่ง BRANCH แทนการกระโดดแบบสัมพัทธ์ได้โดยจะต้องเลือกเงื่อนไขให้ถูกต้อง คือ ไม่มีกรณีของเงื่อนไขเป็นเท็จ เกิดขึ้นได้เลย

2. การกระโดดด้วยคำสั่ง RTS คุณคงคุ้นเคยแต่การใช้คำสั่ง RTS ควบคู่กับคำสั่ง JSR มามาก จนกระทั่งลืมไปว่าคำสั่ง RTS ก็สามารถทำให้เกิดการกระโดดของโปรแกรมได้เช่นกัน จะกระโดดได้อย่างไรรึ...

ด้วยคำสั่ง RTS นี้ จะทำให้ซีพียูนำค่าแอดเดรสที่เก็บไว้ในสแต็คมาใส่คืนในโปรแกรมเคาน์เตอร์ ดังนั้นในการกระโดดโดยใช้คำสั่ง RTS เราจะต้องเริ่มด้วยการนำค่าแอดเดรสที่จะกระโดดไปนั้น เก็บลงไปในสแต็คให้แอดเดรสไบท์สูงไปก่อน แล้วจึงใส่ค่าแอดเดรสไบท์ต่ำทับลงไป จากนั้นก็เริ่มกระโดดด้วยคำสั่ง RTS ได้เลย ดังโปรแกรมตัวอย่างในรูปที่ 17 นี้

17

```
LDA    HIBYTE
PHA
LDA    LOBYTE
PHA
RTS
```

อ้อ! ผมเกือบจะลืมเตือนอะไรไปสักอย่างหนึ่งเกี่ยวกับคำสั่ง RTS ที่ว่า ถ้ามีการนำค่าแอดเดรสจากสแต็คมาใส่ไว้ในโปรแกรมเคาน์เตอร์แล้ว จะมีการเพิ่มค่าในโปรแกรมเคาน์เตอร์ขึ้นอีกหนึ่งโดยอัตโนมัติ ดังนั้นถ้าคุณต้องการกระโดดไปยังแอดเดรส $6005 ก็จะต้องพุชค่า $60 ลงไปในสแต็คก่อน แล้วจึงพุชค่า $04 ตามลงไปในสแต็ค จึงจะทำให้มีการกระโดดไปยังแอดเดรสที่คุณต้องการ

การกระโดดในลักษณะเช่นนี้ คงจะไม่เหมาะกับการกระโดดไปยังแอดเดรสคงที่เหมือนกับคำสั่ง JMP ทั่วไป แต่สามารถจะใช้ได้ดีกับการกระโดดที่ต้องอาศัย JUMP TABLE เพื่อให้สามารถกระโดดไปยังแอดเดรสต่างๆ ได้หลายที่ โดยสามารถนำมาใช้ทดแทนคำสั่ง JMP (ADR) ได้เป็นอย่างดี

คุณสามารถจะดูตัวอย่างได้จากโปรแกรมมอนิเตอร์ของเครื่องแอปเปิ้ลทูในช่วงแอดเดรส FF78-FFF9 ซึ่งถ้าดูจากหนังสือคู่มือของเครื่องแอปเปิ้ลทูจะสามารถเข้าใจได้ง่ายขึ้น

เขียนมาถึงตอนนี้ ผมก็จำเป็นจะต้องห้ามมือของตัวเองเอาไว้ ก่อนที่จะเขียนหัวข้ออื่นๆ ต่อไปอีก เพราะบทความคงจะไม่จบแน่ๆ เลย ถ้าหากปล่อยให้เป็นอย่างนั้น(ขอโทษทีเถอะ! โม้มากไปหน่อย) คุณๆ ก็อย่าเพิ่งคิดว่าภาษาแอสเซมบลีเป็นภาษาที่ยุ่งยากวุ่นวาย เลยพาลไม่อยากเขียนเอาเสียเลยนะครับ แต่ว่าที่ผมเขียนให้อ่านเยอะๆ นี้ ก็เพื่อเพิ่มความสามารถในการเขียนโปรแกรมแอสเซมบลีของคุณ คุณคงจะรู้สึกเหมือนกับผมว่ายิ่งคุณได้รู้รายละเอียดมากๆ ขึ้น ก็จะยิ่งเพิ่มอำนาจในการบังคับเครื่องคอมพิวเตอร์ได้มากขึ้นด้วย จริงไหม!